Reparametrizing Shampoo and SOAP for Subspace Basis Updates and BFloat16 Storage

author names withheld

Under Review for the Workshop on High-dimensional Learning Dynamics, 2026

Abstract

Shampoo-based methods, such as KL-Shampoo and SOAP, have demonstrated strong performance in training neural networks and leverage QR decompositions. As existing QR implementations require single-precision arithmetic and remain computationally expensive, these methods become time- and memory-intensive when their preconditioning matrices are large. Moreover, using half-precision (BFP16) storage to reduce memory can degrade the performance of Shampoo-based methods. We propose a reparametrization of the preconditioner that supports half-precision storage, and also enables efficient QR-based updates in subspaces while retaining single-precision arithmetic and thereby reducing both computational cost and memory overhead. It applies broadly to Shampoo-based methods that employ QR decomposition, including KL-Shampoo and SOAP. Our approach mitigates the performance degradation of these methods under half-precision storage and, overall, makes them more memory- and time-efficient.

1. Introduction

Shampoo-based methods, such as Shampoo [11, 23], SOAP [25], and KL-Shampoo/SOAP [16], have recently attracted considerable attention because of their strong performance in training neural networks [6, 8, 15]. They employ non-diagonal preconditioners and, for a matrix-valued weight, use a Kronecker-factored preconditioner together with its inverse matrix root for preconditioning. To compute this root, pure Shampoo-based methods often perform eigendecompositions of each Kronecker factor, $\boldsymbol{S}_i = \boldsymbol{Q}_i \mathrm{Diag}(\boldsymbol{\lambda}_i) \boldsymbol{Q}_i^\top$, and store $\boldsymbol{\lambda}_i, \boldsymbol{Q}_i, \boldsymbol{S}_i$ [7, 23]. SOAP-type methods also rely on eigendecomposition because they run Adam in the eigenbasis $\boldsymbol{Q}_i$. However, eigendecomposition becomes expensive for large matrices, and existing implementations require single-precision arithmetic (e.g., LAPACK [2] on CPUs and cuSOLVER/MAGMA [13, 19] on GPUs, as used in JAX [5] and PyTorch [20]). As a result, preconditioning factors such as $\boldsymbol{Q}_i$ and $\boldsymbol{S}_i$ are typically stored in single precision [3, 7, 23]. More recently, QR decomposition has been proposed as a cheaper approximation to eigendecomposition, reducing computational cost while retaining the same arithmetic [7, 16, 25]. Still, QR decomposition remains the main computational bottleneck and is expensive relative to other subroutines (e.g., matrix multiplications; see fig. 1, right). One possible way to reduce this cost is to consider subspace updates. However, it remains unclear how to do so using the current optimizer state.

In this work, we propose a reparametrization of the preconditioning factors in Shampoo-based methods to address these limitations. Specifically, for each Kronecker factor $\boldsymbol{S}_i$, we store $\{\boldsymbol{\lambda}_i, \boldsymbol{Q}_i, \boldsymbol{P}_i\}$ instead of $\{\boldsymbol{\lambda}_i, \boldsymbol{Q}_i, \boldsymbol{S}_i\}$, where $\boldsymbol{P}_i := \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i$, and we directly update $\boldsymbol{P}_i$ without materializing $\boldsymbol{S}_i$:

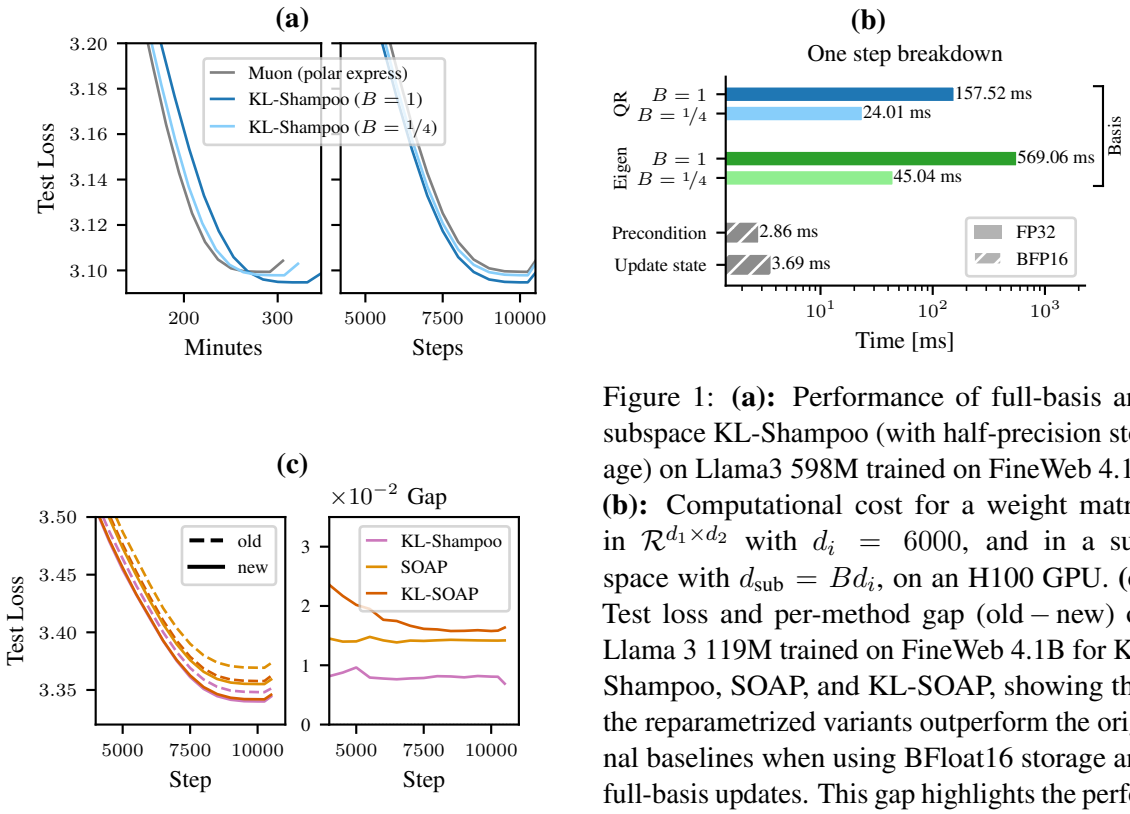

Figure 1: **(a):** Performance of full-basis and subspace KL-Shampoo (with half-precision storage) on Llama3 598M trained on FineWeb 4.1B. **(b):** Computational cost for a weight matrix in $\mathcal{R}^{d_1 \times d_2}$ with $d_i = 6000$, and in a subspace with $d_{\text{sub}} = Bd_i$, on an H100 GPU. **(c):** Test loss and per-method gap (old – new) on Llama 3 119M trained on FineWeb 4.1B for KL-Shampoo, SOAP, and KL-SOAP, showing that the reparametrized variants outperform the original baselines when using BFloat16 storage and full-basis updates. This gap highlights the performance degradation caused by BFloat16 storage under the original parametrization.

- This enables efficient updates of a subset of the (orthogonal) basis vectors in $\boldsymbol{Q}_i$ via QR in subspaces of $\boldsymbol{P}_i$, thereby significantly reducing the cost of QR (see the top right of fig. 1) while supporting memory-efficient half-precision storage.
- It is compatible with a variety of subspace selection strategies, including random selection and Jacobi-style selection [24, 26] originally designed for subspace eigendecomposition. This approach applies broadly to Shampoo-based methods that employ QR decomposition, including KL-Shampoo, SOAP, and KL-SOAP.
- Empirically, our approach improves the performance of SOAP-type methods and closes the performance gap between KL-Shampoo and KL-SOAP reported by Lin et al. [16] (see bottom left of fig. 1 and section 3), even with full-basis QR updates, when all preconditioning factors are stored in half precision. Our approach also narrows the runtime gap between KL-Shampoo and Muon and opens new directions to further improvement. (see section E).

2. Reparametrization for Reducing Time and Memory Costs

For a detailed discussion regarding existing Shampoo based methods and their drawbacks, we refer to section A. We propose to reduce time and memory costs by reparametrizing the preconditioning factors $\boldsymbol{S}_i$ as $\boldsymbol{P}_i = \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i$. This reparametrization enables Shampoo-based methods to support both full-basis and subspace orthogonal updates, as well as half-precision storage, while empirically

mitigating performance degradation. As we will show, these updates require performing QR decomposition on a subspace of $\boldsymbol{P}_i$ (see fig. 3). This motivates directly storing and using $\boldsymbol{P}_i$ as the reparametrization of the preconditioning factor $\boldsymbol{S}_i$.

We begin by modifying an existing update scheme to use this reparametrization. To this end, we establish the mathematical equivalence between the existing parametrization (e.g., $\boldsymbol{S}_i$) and our reparametrization (e.g., $\boldsymbol{P}_i$) in the full-basis setting. We then show that, unlike the existing parametrization, our reparametrization not only supports half-precision storage without compromising performance in the full-basis setting, but also enables efficient subspace orthogonal updates (e.g., updates of a subset of basis vectors in $\boldsymbol{Q}_i$ via QR on a subspace of $\boldsymbol{P}_i$). Notably, we keep the updates of the eigenvalues (e.g., $\boldsymbol{\lambda}_i$ for pure Shampoo-based methods or $\boldsymbol{d}$ for SOAP-type methods) and preconditioning factors (e.g., $\boldsymbol{P}_i$) in the full basis (see Step 2 & 3 in fig. 2). This is essential for mitigating performance degradation of these methods under subspace updates while reducing the cost of QR decomposition, the main computational bottleneck (see the top-right panel of fig. 1).

2.1. Full-basis Update: Memory Reduction via Half-precision Storage

For simplicity, we assume in this section that a QR implementation returns a unique decomposition. See section B for making any QR implementation unique.

Recall that existing QR-based Shampoo methods [16, 25] store $\boldsymbol{S}_i$ and compute $\boldsymbol{Q}_i^{(\text{new})}$ via a full-basis QR decomposition $\boldsymbol{Q}_i^{(\text{new})}\boldsymbol{R}_i^{(\text{new})} = \boldsymbol{S}_i\boldsymbol{Q}_i^{(\text{old})}$ to approximate an orthogonal eigenbasis of $\boldsymbol{S}_i$, for $i \in \{1, 2\}$, For example, consider the KL-Shampoo update in fig. 2.

Mathematical equivalence for deriving our update scheme Modifying the existing QR-based update scheme for our reparametrization builds on the observation that performing QR decomposition on the reparametrized matrix $\boldsymbol{P}_i^{(\text{old})} := \boldsymbol{Q}_i^{(\text{old})\top}\boldsymbol{S}_i\boldsymbol{Q}_i^{(\text{old})}$ yields

$$\boldsymbol{O}_i\bar{\boldsymbol{R}}_i = \boldsymbol{P}_i^{(\text{old})} = \boldsymbol{Q}_i^{(\text{old})\top}\left[\boldsymbol{S}_i\boldsymbol{Q}_i^{(\text{old})}\right] = \boldsymbol{Q}_i^{(\text{old})\top}\left[\boldsymbol{Q}_i^{(\text{new})}\boldsymbol{R}_i^{(\text{new})}\right] = \left[\boldsymbol{Q}_i^{(\text{old})\top}\boldsymbol{Q}_i^{(\text{new})}\right]\boldsymbol{R}_i^{(\text{new})} \tag{1}$$

We then use the equivalence established in eq. (1) to obtain a new update rule for computing $\boldsymbol{Q}_i$ under our reparametrization. Notice that both the left-hand side and the right-hand side of eq. (1) are QR decompositions of $\boldsymbol{P}_i^{(\text{old})}$. Because the QR decomposition for non-singular $\boldsymbol{P}_i^{(\text{old})}$ is unique under our assumption, we obtain the relationship

$$\boldsymbol{O}_i = \boldsymbol{Q}_i^{(\text{old})\top}\boldsymbol{Q}_i^{(\text{new})}, \qquad \bar{\boldsymbol{R}}_i = \boldsymbol{R}_i^{(\text{new})}.$$

This implies that $\boldsymbol{Q}_i^{(\text{new})} = \boldsymbol{Q}_i^{(\text{old})}\boldsymbol{O}_i$ because $\boldsymbol{Q}_i^{(\text{old})}\boldsymbol{Q}_i^{(\text{old})\top} = \boldsymbol{I}$. This relationship leads to the following update scheme for $\boldsymbol{Q}_i$ under our reparametrization: perform QR decomposition on $\boldsymbol{P}_i^{(\text{old})}$ to obtain $\boldsymbol{O}_i$, and then update $\boldsymbol{Q}_i$ via

$$\boldsymbol{Q}_i^{(\text{new})} = \boldsymbol{Q}_i^{(\text{old})}\boldsymbol{O}_i. \tag{2}$$

Importantly, this scheme enables efficient subspace update schemes, as discussed in section 2.2.

Since $\boldsymbol{P}_i$ depends on $\boldsymbol{Q}_i$, we need to update $\boldsymbol{P}_i$ via

$$\boldsymbol{P}_i^{(\text{new})} := \boldsymbol{Q}_i^{(\text{new})\top}\boldsymbol{S}_i\boldsymbol{Q}_i^{(\text{new})} = \boldsymbol{O}_i^\top\boldsymbol{Q}_i^{(\text{old})\top}\boldsymbol{S}_i\boldsymbol{Q}_i^{(\text{old})}\boldsymbol{O}_i = \boldsymbol{O}_i^\top\boldsymbol{P}_i^{(\text{old})}\boldsymbol{O}_i, \tag{3}$$

which avoids explicitly forming $\boldsymbol{S}_i$ when the basis is changed from $\boldsymbol{Q}_i^{(\text{old})}$ to $\boldsymbol{Q}_i^{(\text{new})}$.

If $\boldsymbol{Q}_i$ is held fixed while $\boldsymbol{S}_i$ is updated via an exponential moving average (EMA) with Δ_i shown in Step 2 in fig. 2, then $\boldsymbol{P}_i$ is equivalently updated using $\tilde{\Delta}_i := \boldsymbol{Q}_i^\top\Delta_i\boldsymbol{Q}_i$:

$$\boldsymbol{Q}_i^\top\boldsymbol{S}_i^{(\text{new})}\boldsymbol{Q}_i = (1-\beta_2)\boldsymbol{Q}_i^\top\boldsymbol{S}_i^{(\text{old})}\boldsymbol{Q}_i + \beta_2\boldsymbol{Q}_i^\top\Delta_i\boldsymbol{Q}_i \iff \boldsymbol{P}_i^{(\text{new})} = (1-\beta_2)\boldsymbol{P}_i^{(\text{old})} + \beta_2\tilde{\Delta}_i. \tag{4}$$

Method	Parameterization	nanoGPT (123M) FP32 → BFP16	llama3 (119M) FP32 → BFP16
KL-Shampoo	old	$3.244 \xrightarrow{+0.006} 3.250$	$\mathbf{3.345} \xrightarrow{+0.000} 3.345$
	new	$3.245 \xrightarrow{+0.001} 3.246$	$\mathbf{3.345} \xrightarrow{-0.001} \mathbf{3.344}$
KL-SOAP	old	$3.240 \xrightarrow{+0.017} 3.257$	$3.351 \xrightarrow{+0.011} 3.362$
	new	$\mathbf{3.237} \xrightarrow{+0.007} \mathbf{3.244}$	$3.346 \xrightarrow{+0.000} 3.346$
SOAP	old	$3.248 \xrightarrow{+0.014} 3.262$	$3.358 \xrightarrow{+0.014} 3.372$
	new	$3.250 \xrightarrow{+0.000} 3.250$	$3.353 \xrightarrow{+0.006} 3.359$

Table 1: Performance of each method in terms of test loss. Each cell shows the result with FP32 storage followed by BFP16 storage; the number above the arrow is the (BFP16−FP32) delta. Small deltas indicate the parameterization is robust to half-precision storage.

Generalization to other Shampoo-based methods Although we describe the changes for KL-Shampoo in fig. 2, our approach applies directly to many other Shampoo-based methods that use QR decomposition. Pure Shampoo-based methods often differ in how they compute Δ_i in Step 2 of fig. 2. Thus, our approach also applies to them. SOAP-type methods do not require Step 3, but they do require a further modification of Step 5 in fig. 2 to run Adam in the basis $\boldsymbol{Q}_i$, which requires only knowledge of $\boldsymbol{Q}_i$ rather than $\boldsymbol{S}_i$. Therefore, our approach also applies to SOAP-type methods.

Our reparametrization supports half-precision storage While the existing parametrization and our reparametrization are mathematically equivalent, our empirical results (see section 3) show that our reparametrization, unlike the old parametrization, preserves the performance of Shampoo-based methods under half-precision storage, thereby enabling half-precision storage without compromising performance. Moreover, $\boldsymbol{P}_i$ is often close to diagonal, which can to further reduce memory usage.

2.2. Subspace Update: Computational Cost Reduction via QR Decomposition in Subspaces

Our update scheme under the reparametrization directly supports updating a subset of the orthogonal basis vectors in $\boldsymbol{Q}_i$ via QR in subspaces of $\boldsymbol{P}_i$. Interestingly, the scheme in eqs. (2) and (3) resembles block Jacobi-type methods [24, 26] for subspace approximation via eigendecomposition. Thus, our reparametrization supports efficient subspace updates via either QR or eigendecomposition. In this work, however, we focus on a QR-based local orthogonal factor within a selected subspace because subspace QR is computationally cheaper (see the top right of fig. 1) while empirically achieving performance similar to that of subspace eigendecomposition for Shampoo-based methods (see table 5). We discuss how this parameterization enables the use of subspaces in section C.

3. Experiments

We conduct five experiments to demonstrate the benefits of our reparametrization. In Experiment 1, we consider the Shampoo-based methods KL-Shampoo, SOAP, and KL-SOAP. The remaining experiments focus on KL-Shampoo as a representative Shampoo-based method because of limited computational resources. Our subspace approach directly applies to SOAP and KL-SOAP, too. See section D for additional experimental details and discussions of Experiments 4 and 5.

Experiment 1: Our reparametrization preserves performance under BFloat16 storage In this set of experiments (see section 3), we show that our reparametrization preserves the performance of Shampoo-based methods when switching the storage format from FP32 to BFP16, even in the full-basis setting. The existing parametrization does not maintain performance under BFP16 storage.

Table 2: This shows that performing multi-subspace updates in KL-Shampoo (ours, BFP16, QR, greedy) can improve accuracy at the cost of increased runtime. We update the basis via multi-step procedures in subspaces. We consider different subspace sizes Cell colors encode the wall-clock runtime ratio relative to the full-basis QR baseline ; see colorbar below.

T	B	K	nanoGPT (123M)	llama3 (119M)	llama3 (313M)
10	1	1	3.245	3.345	3.183
10	1/2	1	3.247	3.344	3.187
		3	3.245	3.344	3.187
		5	3.244	3.345	3.184
10	1/4	1	3.251	3.347	3.192
		3	3.251	3.346	3.189
		5	3.248	3.345	3.184
10	1/6	1	3.256	3.346	3.195
		3	3.251	3.346	3.189
		5	3.252	3.346	3.187

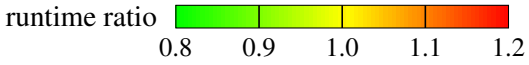

Table 3: This shows that updating the KL-Shampoo basis (ours, BFP16, QR, greedy) more frequently ($T = 2, 3, 4, 5$) can be beneficial even when updating only a subset of the basis ($d_{\text{sub},i} = Bd_i$). A cheap subspace update can be useful when the basis is updated more frequently.

T	B	K	nanoGPT (123M)	llama3 (119M)	llama3 (313M)
2	1/3	1	3.247	3.344	3.183
	1/4		3.250	3.345	3.184
	1/5		3.252	3.345	3.184
3	1/3	1	3.246	3.344	3.184
	1/4		3.251	3.345	3.184
	1/5		3.253	3.346	3.186
4	1/3	1	3.247	3.345	3.184
	1/4		3.252	3.346	3.188
	1/5		3.250	3.346	3.190
5	1/3	1	3.249	3.345	3.186
	1/4		3.252	3.347	3.187
	1/5		3.254	3.347	3.190

According to section 3, our reparametrization also closes the performance gap between KL-Shampoo and KL-SOAP reported by Lin et al. [16] when BFP16 storage is used (see bottom-left of fig. 1).

Experiment 2: Our reparametrization supports subspace selection methods This set of experiments (see table 4) illustrates the use of subspace selection methods with a K-step inner loop. A straightforward block-selection strategy is random selection via uniform sampling. Thus, we consider both this random strategy and the greedy method described in section 2.2. From table 4, we can see that the greedy selection is generally more effective than random selection. On nanoGPT, the greedy method performs much better because of nanoGPT's aggressive learning schedule. Based on these results, we focus on the greedy selection method in the remaining experiments.

Experiment 3: Our reparametrization supports efficient subspace QR updates Block Jacobi methods often employ an inner loop and perform multi-step iterations (e.g., $K = 1, 3, 5$) in We investigate whether using the same loop is necessary for Shampoo-based methods. **(I) With the inner loop:** We begin by fixing the decomposition frequency for both full-basis and subspace updates. As shown in table 2, subspace updates can achieve performance comparable to that of full-basis updates when using the same loop given a sufficient subspace size. Using multi-step subspace updates can improve accuracy at the cost of increased runtime. By contrast, using a single step is inexpensive but inaccurate. **(II) Without the inner loop:** Unlike the classical setting, $\boldsymbol{P}_i$ is ever-changing rather than fixed. This motivates us to vary the decomposition frequency and use a single subspace update (i.e., $K = 1$). This has a similar spirit to using QR decomposition as an approximation to eigendecomposition. Our experiments show (see table 3) that a single subspace update can achieve decent performance relative to the full-basis method while reducing the total runtime. This finding highlights an accuracy-performance trade-off for improving Shampoo-based methods.

References

[1] Noah Amsel, David Persson, Christopher Musco, and Robert M Gower. The polar express: Optimal matrix sign methods and their application to the Muon algorithm. In *International Conference on Learning Representations (ICLR)*, 2026.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. 1999.

[3] Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020.

[4] Martin Bečka, Gabriel Okša, and Marian Vajteršic. Dynamic ordering for a parallel block-Jacobi SVD algorithm. *Parallel Computing*, 2002.

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018.

[6] George E Dahl, Frank Schneider, Zachary Nado, Naman Agarwal, Chandramouli Shama Sastry, Philipp Hennig, Sourabh Medapati, Runa Eschenhagen, Priya Kasimbeg, Daniel Suo, et al. Benchmarking neural network training algorithms, 2023.

[7] Runa Eschenhagen, Aaron Defazio, Tsung-Hsien Lee, Richard E Turner, and Hao-Jun Michael Shi. Purifying Shampoo: Investigating Shampoo's heuristics by decomposing its preconditioner. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[8] Runa Eschenhagen, Anna Cai, Tsung-Hsien Lee, and Hao-Jun Michael Shi. Clarifying Shampoo: Adapting spectral descent to stochasticity and the parameter trajectory. *arXiv preprint arXiv:2602.09314*, 2026.

[9] George Elmer Forsythe and Peter Henrici. The cyclic Jacobi method for computing the principal values of a complex matrix. *Transactions of the American Mathematical Society*, 1960.

[10] Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU press, 2013.

[11] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning (ICML)*, 2018.

[12] Nicholas J Higham and Theo Mary. Mixed precision algorithms in numerical linear algebra. *Acta Numerica*, 2022.

[13] Innovative Computing Laboratory. MAGMA: Matrix algebra on GPU and multicore architectures, 2026. URL https://developer.nvidia.com/magma. Accessed: 2026-05-06.

[14] Keller Jordan. NanoGPT (124M) in 3 minutes. https://github.com/KellerJordan/modded-nanogpt, 2024. Accessed: 2025-06.

[15] Priya Kasimbeg, Frank Schneider, Runa Eschenhagen, Juhan Bae, Chandramouli Shama Sastry, Mark Saroufim, Boyuan Fend, Less Wright, Edward Z Yang, Zachary Nado, et al. Accelerating neural network training: An analysis of the AlgoPerf competition. In *The Thirteenth International Conference on Learning Representations*, 2025.

[16] Wu Lin, Scott C. Lowe, Felix Dangel, Runa Eschenhagen, Zikun Xu, and Roger B. Grosse. Understanding and improving Shampoo and SOAP via Kullback-Leibler minimization. In *International Conference on Learning Representations (ICLR)*, 2026.

[17] Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, Yanru Chen, Huabin Zheng, Yibo Liu, Shaowei Liu, Bohong Yin, Weiran He, Han Zhu, Yuzhi Wang, Jianzhou Wang, Mengnan Dong, Zheng Zhang, Yongsheng Kang, Hao Zhang, Xinran Xu, Yutao Zhang, Yuxin Wu, Xinyu Zhou, and Zhilin Yang. Muon is scalable for llm training. *arXiv*, 2025.

[18] Depen Morwani, Itai Shapira, Nikhil Vyas, Eran Malach, Sham Kakade, and Lucas Janson. A new perspective on Shampoo's preconditioner. In *International Conference on Learning Representations (ICLR)*, 2024.

[19] NVIDIA Corporation. *cuSOLVER Library*, 2026. URL https://docs.nvidia.com/cuda/cusolver/index.html. Accessed: 2026-05-06.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

[21] Sebastian Raschka. Build a large language model (from scratch). https://github.com/rasbt/LLMs-from-scratch, 2024. Accessed: 2025-10.

[22] Andrei Semenov, Matteo Pagliardini, and Martin Jaggi. Benchmarking optimizers for large language model pretraining. *arXiv preprint arXiv:2509.01440*, 2025.

[23] Hao-Jun Michael Shi, Tsung-Hsien Lee, Shintaro Iwasaki, Jose Gallego-Posada, Zhijing Li, Kaushik Rangadurai, Dheevatsa Mudigere, and Michael Rabbat. A distributed data-parallel PyTorch implementation of the distributed Shampoo optimizer for training neural networks at-scale, 2023.

[24] Charles Van Loan. The block Jacobi method for computing the singular value decomposition. Technical report, Cornell University, 1985.

[25] Nikhil Vyas, Depen Morwani, Rosie Zhao, Itai Shapira, David Brandfonbrener, Lucas Janson, and Sham M Kakade. SOAP: Improving and stabilizing Shampoo using Adam for language modeling. In *International Conference on Learning Representations (ICLR)*, 2025.

[26] Yusaku Yamamoto, Zhang Lan, and Shuhei Kudo. Convergence analysis of the parallel classical block Jacobi method for the symmetric eigenvalue problem. *JSIAM Letters*, 2014.

Appendix A. Background

Notation Because each weight matrix is treated independently in Shampoo-based methods, we focus on a single weight matrix, denoted by $\boldsymbol{\Theta} \in \mathbb{R}^{d_1 \times d_2}$, rather than on all weight matrices in neural network optimization, in order to simplify the notation. To further simplify the discussion, we omit weight decay and momentum, which are also used in Shampoo-based methods. Here, $\boldsymbol{G}$ denotes the gradient matrix with respect to $\boldsymbol{\Theta}$, and $\boldsymbol{g} := \mathrm{vec}(\boldsymbol{G}) \in \mathbb{R}^{d_1 d_2 \times 1}$ denotes the flattened form of $\boldsymbol{G}$. We use $\mathrm{diag}(\cdot)$ to extract the diagonal entries of an input matrix, while $\mathrm{Diag}(\cdot)$ denotes the diagonal matrix whose diagonal entries are given by the input vector.

Pure Shampoo-based methods These methods, such as Shampoo [7, 18, 23] and KL-Shampoo [16], employ a Kronecker-factorized preconditioning matrix $\boldsymbol{S}_{\text{shampoo}} = \boldsymbol{S}_1 \otimes \boldsymbol{S}_2$ for each matrix-valued weight $\boldsymbol{\Theta}$ and perform the following preconditioning step using an inverse matrix square root:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \boldsymbol{S}_{\text{shampoo}}^{-1/2} \boldsymbol{g} \iff \boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \gamma \boldsymbol{S}_1^{-1/2} \boldsymbol{G} \boldsymbol{S}_2^{-1/2},$$

where $\boldsymbol{\theta} := \mathrm{vec}(\boldsymbol{\Theta})$, $\otimes$ denotes the Kronecker product, $\gamma > 0$ is the step size, $\boldsymbol{S}_i \in \mathbb{R}^{d_i \times d_i}$ is a Kronecker factor matrix, and the equivalence follows from properties of the Kronecker product.

Eigendecomposition simplifies the computation of the matrix root Many existing works [7, 23] suggest performing an eigendecomposition of each Kronecker factor, $\boldsymbol{S}_i = \boldsymbol{Q}_i \mathrm{Diag}(\boldsymbol{\lambda}_i) \boldsymbol{Q}_i^\top$ for $i \in \{1, 2\}$, in order to compute the inverse square root. This yields

$$\boldsymbol{S}_{\text{shampoo}}^{-1/2} = (\boldsymbol{Q}_1 \mathrm{Diag}(\boldsymbol{\lambda}_1^{\odot -1/2}) \boldsymbol{Q}_1^\top) \otimes (\boldsymbol{Q}_2 \mathrm{Diag}(\boldsymbol{\lambda}_2^{\odot -1/2}) \boldsymbol{Q}_2^\top) = \boldsymbol{Q} \mathrm{Diag}(\boldsymbol{\lambda}_1^{\odot -1/2} \otimes \boldsymbol{\lambda}_2^{\odot -1/2}) \boldsymbol{Q}^\top,$$

where $\odot$ denotes an elementwise operation, $\boldsymbol{Q}_i$ is the orthogonal factor obtained from the decomposition, and $\boldsymbol{Q} := \boldsymbol{Q}_1 \otimes \boldsymbol{Q}_2$ is the eigenbasis of $\boldsymbol{S}_{\text{shampoo}}$.

SOAP-type methods Methods such as SOAP [25] and KL-SOAP [16] require eigendecomposition or its approximations because they employ an augmented matrix $\boldsymbol{S}_{\text{soap}} := \boldsymbol{Q} \mathrm{Diag}(\boldsymbol{d}) \boldsymbol{Q}^\top$ for preconditioning, where $\boldsymbol{d} \in \mathbb{R}^{d_1 d_2 \times 1}$ is known as Adam's second-moment vector in the basis $\boldsymbol{Q}$. Notably, $\boldsymbol{d}$ cannot be expressed as a Kronecker product, such as $\boldsymbol{\lambda}_1 \otimes \boldsymbol{\lambda}_2$, as in pure Shampoo-based methods [16]. Thus, a SOAP-type method requires eigendecomposition or its approximations in order to estimate $\boldsymbol{d}$.

Using stale eigenbasis can reduce the runtime cost with the price of performance degradation Many existing works perform eigendecomposition infrequently because of its high computational cost (see fig. 1). The eigenbasis $\boldsymbol{Q}$ obtained from the most recent decomposition is stored and reused for preconditioning in order to avoid performing the decomposition at every iteration. To make Shampoo-based methods competitive, the decomposition interval T must be large (e.g., performing the decomposition every $T = 50$ gradient steps). However, a large decomposition interval can significantly degrade the performance of Shampoo-based methods [7, 23, 25] because of the staleness of the eigenbasis. Thus, we cannot simply reduce the cost of this decomposition by increasing the interval without sacrificing performance.

QR decomposition is a cheaper approximation to improve runtime and reduce performance degradation Motivated by the trade-off between performance and computational cost, QR decomposition has been proposed as a cheaper approximation to eigendecomposition [7, 16, 25]:

$\boldsymbol{Q}_i^{(\text{new})} = \text{qr}(\boldsymbol{S}_i \boldsymbol{Q}_i^{(\text{old})})$ for $i \in \{1, 2\}$, to reduce computational cost (see fig. 1) while maintaining the performance of Shampoo-based methods, where $\boldsymbol{Q}_i^{(\text{old})}$ is a stale eigenbasis. This is possible because QR decomposition allows a smaller decomposition interval (e.g., $T = 10$) to improve the performance while keeping the cost low.

Tracking eigenvalues is essential when using stale bases Because QR decomposition can only approximate eigenbases, existing works propose tracking eigenvalues separately, for example, through estimation schemes for pure Shampoo-based methods [16] and SOAP-type methods [25]. Eschenhagen et al. [7], Lin et al. [16] further show that updating eigenvalues at each step is essential when using outdated bases $\boldsymbol{Q}_i$ to reduce runtime cost. We will show our reparametrization is compatible with these schemes (see Step 3 of fig. 2).

A.1. Limitations induced by matrix decomposition

As we discussed before, matrix decomposition is required by SOAP-type methods and recommended for pure Shampoo-based methods. However, there are limitations in using matrix decomposition that must be overcome to further unlock the potential of these methods for neural network optimization.

Half-precision may degrade performance, whereas single precision increases memory consumption Existing implementations of eigendecomposition and QR decomposition in LAPACK and cuSOLVER/MAGMA, as used in JAX and PyTorch, require single-precision arithmetic in order to remain numerically stable. This is because these algorithms are typically studied and analyzed in single-precision and, in some cases, double-precision settings [10]. Decomposition algorithms that support half-precision computation are not yet widely available and remain under active development [12]. As a result, preconditioning factor matrices are often stored in full precision [3, 23]. Unfortunately, this increases memory consumption. Although mixed-precision schemes [16, 25], such as combining half-precision storage with full-precision decomposition, are possible, half-precision storage can degrade the performance of Shampoo-based methods. For example, Lin et al. [16] report that KL-SOAP underperforms KL-Shampoo when half-precision storage is used.

Matrix decomposition is the dominant computational cost Matrix decomposition algorithms such as eigen- and QR decomposition have cubic complexity and, unlike matrix multiplication, they require full-precision arithmetic and are difficult to parallelize on GPUs. Figure 1 shows that matrix decompositions are main computational bottleneck in Shampoo-based methods. Thus, making QR decomposition faster can make these methods more competitive.

Appendix B. Make any QR Implementation Unique

For a non-singular square matrix $\boldsymbol{A}$, we can always make any QR implementation unique by requiring the upper-triangular matrix $\boldsymbol{R}$ to have *positive* diagonal entries. This is possible because $\boldsymbol{R}$ must have non-zero diagonal entries when $\boldsymbol{A}$ is non-singular. We then use the following procedure to make the decomposition unique

$$\text{qr}(\boldsymbol{A}) = \underbrace{\boldsymbol{Q}\boldsymbol{R}}_{\text{non-unique QR}} = \underbrace{\boldsymbol{Q}\boldsymbol{D}}_{=\bar{\boldsymbol{Q}}} \underbrace{\boldsymbol{D}^\top \boldsymbol{R}}_{=\bar{\boldsymbol{R}}}$$

KL-Shampoo (Old)	**KL-Shampoo (Reparameterized)**
Optimizer state $\{\boldsymbol{\lambda}_i, \boldsymbol{Q}_i, \boldsymbol{S}_i \mid i = 1, 2\}$	Optimizer state $\{\boldsymbol{\lambda}_i, \boldsymbol{Q}_i, \boldsymbol{P}_i := \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i \mid i = 1, 2\}$
1: Compute gradient $\boldsymbol{g} := \nabla \ell(\boldsymbol{\theta})$, $\boldsymbol{G} := \mathrm{Mat}(\boldsymbol{g}) \in \mathbb{R}^{d_1 \times d_2}$	
2: Estimate preconditioning factors (each iteration, full-basis)	
	$\tilde{\boldsymbol{G}}' := \boldsymbol{Q}_1^\top \boldsymbol{G} \boldsymbol{Q}_2$ # 2 mms
$\boldsymbol{G}'_1 := \boldsymbol{G}\boldsymbol{Q}_2 \,\mathrm{Diag}(\boldsymbol{\lambda}_2^{-1/2})/\sqrt{d_2}$ # 1 mm	$\tilde{\boldsymbol{G}}'_1 := \tilde{\boldsymbol{G}}' \,\mathrm{Diag}(\boldsymbol{\lambda}_2^{-1/2})/\sqrt{d_2}$
$\boldsymbol{G}'_2 := \boldsymbol{G}^\top \boldsymbol{Q}_1 \,\mathrm{Diag}(\boldsymbol{\lambda}_1^{-1/2})/\sqrt{d_1}$ # 1 mm	$\tilde{\boldsymbol{G}}'_2 := \tilde{\boldsymbol{G}}'^\top \,\mathrm{Diag}(\boldsymbol{\lambda}_1^{-1/2})/\sqrt{d_1}$
$\Delta_i := \boldsymbol{G}'_i \boldsymbol{G}'^\top_i$ # 2 mms	$\tilde{\Delta}_i := \tilde{\boldsymbol{G}}'_i \tilde{\boldsymbol{G}}'^\top_i \equiv \boldsymbol{Q}_i^\top \Delta_i \boldsymbol{Q}_i$ # 2 mms
$\boldsymbol{S}_i \leftarrow (1-\beta_2)\boldsymbol{S}_i + \beta_2 \Delta_i$	$\boldsymbol{P}_i \leftarrow (1-\beta_2)\boldsymbol{P}_i + \beta_2 \tilde{\Delta}_i$
3: Track eigenvalues via EMA (each iteration, full-basis)	
$\tilde{\boldsymbol{G}}'_i := \boldsymbol{Q}_i^\top \boldsymbol{G}'_i$ # 2 mms $\mathrm{diag}(\tilde{\Delta}_i) = (\tilde{\boldsymbol{G}}'_i \odot \tilde{\boldsymbol{G}}'_i)\mathbf{1}$	Use $\tilde{\Delta}_i$ for free
$\boldsymbol{\lambda}_i \leftarrow (1-\beta_2)\boldsymbol{\lambda}_i + \beta_2 \,\mathrm{diag}(\tilde{\Delta}_i)$	
4: Estimate eigenbasis via QR (every $T \geq 1$ iterations)	
$\boldsymbol{Z}_i \leftarrow \boldsymbol{S}_i \boldsymbol{Q}_i$ # 2 mms $\boldsymbol{Q}_i \leftarrow \mathrm{qr}(\boldsymbol{Z}_i)$ # $O(d_i^3)$	$\boldsymbol{O}_i \leftarrow \mathrm{qr}(\boldsymbol{P}_i)$ # $O(d_i^3)$ $\boldsymbol{Q}_i \leftarrow \boldsymbol{Q}_i \boldsymbol{O}_i$ # 2 mms $\boldsymbol{P}_i \leftarrow \boldsymbol{O}_i^\top \boldsymbol{P}_i \boldsymbol{O}_i$ # 4 mms
5: Precondition using $\boldsymbol{Q} := \boldsymbol{Q}_1 \otimes \boldsymbol{Q}_2$ and learning rate γ	
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma(\boldsymbol{Q}\,\mathrm{Diag}(\boldsymbol{\lambda}_1 \otimes \boldsymbol{\lambda}_2)^{-1/2}\boldsymbol{Q}^\top)\boldsymbol{g}$ # $O(d_1^2 d_2 + d_1 d_2^2)$	

Figure 2: **Side-by-side comparison of KL-Shampoo (left) and its reparameterized variant (right).** The reparameterized variant stores $\boldsymbol{P}_i := \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i$ instead of $\boldsymbol{S}_i$; given equivalent initial state, the two variants produce equivalent iterates (see section 2.1), justifying the term *reparameterization*. **Color code.** Symbols in red mark where the two variants differ. Symbols in blue mark the rotated factors $\tilde{\boldsymbol{G}}'_i = \boldsymbol{Q}_i^\top \boldsymbol{G}'_i$: the old variant forms them in step 3 to extract $\mathrm{diag}(\tilde{\Delta}_i)$, while the reparameterized variant obtains them from the shared intermediate $\tilde{\boldsymbol{G}}' = \boldsymbol{Q}_1^\top \boldsymbol{G} \boldsymbol{Q}_2$ in step 2 and reuses them to update $\boldsymbol{P}_i$ — thereby reusing work that is already done. **Cost.** Gray annotations count matrix–matrix products (mm), summed over $i = 1, 2$. The covariance/eigenvalue update (steps 2–3, every iteration) is 2 mm cheaper for the reparameterized variant, whereas the eigenbasis update (step 4, every $T \geq 1$ iterations) costs 4 mm more; the latter amortizes for any $T \geq 2$, so the reparameterization is strictly cheaper in the regime $T \geq 2$ used in practice.

where $\boldsymbol{D}\boldsymbol{D}^\top = \boldsymbol{I}$ and $\boldsymbol{D} := \mathrm{Diag}(\mathrm{sign}(\mathrm{diag}(\boldsymbol{R})))$ is constructed to be a diagonal sign matrix so that $\bar{\boldsymbol{R}} := \boldsymbol{D}\boldsymbol{R}$ has positive diagonal entries. Note that $\bar{\boldsymbol{Q}}$ is orthogonal and $\bar{\boldsymbol{R}}$ is upper-triangular. Thus, we obtain a unique QR decomposition of $\boldsymbol{A}$ via $\boldsymbol{A} = \bar{\boldsymbol{Q}}\bar{\boldsymbol{R}}$.

Appendix C. QR Decomposition in Subspaces

We now describe how our reparametrization directly enables a single subspace update step. For illustration, suppose that $\boldsymbol{P}_i$ is partitioned as

$$\boldsymbol{P}_i = \begin{bmatrix} \boldsymbol{P}_i^{(XX)} & \boldsymbol{P}_i^{(XY)} \\ \boldsymbol{P}_i^{(YX)} & \boldsymbol{P}_i^{(YY)} \end{bmatrix} \quad \text{with } \boldsymbol{P}_i[:, \mathbb{I}_i] = \begin{bmatrix} \boldsymbol{P}_i^{(XX)} \\ \boldsymbol{P}_i^{(YX)} \end{bmatrix},$$

and that the block $\boldsymbol{P}_i^{(XX)} = \boldsymbol{P}_i[\mathbb{I}_i, \mathbb{I}_i]$, highlighted in red, is selected for the update.

Conceptually, we perform QR decomposition only on $\boldsymbol{P}_i^{(XX)}$ to obtain a local orthogonal factor $\boldsymbol{O}_i^{(XX)}$, and then define $\boldsymbol{O}_i = \boldsymbol{O}_i^{(XX)} \oplus \boldsymbol{I}^{(YY)}$, where $\oplus$ represents the direct sum of matrices to form a block diagonal matrix and $\boldsymbol{I}^{(YY)}$ is an identity matrix. We then use this block-diagonal orthogonal matrix in the full-basis update rules shown in eqs. (2) and (3). Substituting this form of $\boldsymbol{O}_i$ into eq. (2) yields the following subspace update $\boldsymbol{Q}_i^{(\text{new})} = \boldsymbol{Q}_i^{(\text{old})}\boldsymbol{O}_i =$

$$\begin{bmatrix} (\boldsymbol{Q}_i^{(\text{old})})^{(XX)} & \underbrace{(\boldsymbol{Q}_i^{(\text{old})})^{(XY)}}_{\text{old basis}} \\ (\boldsymbol{Q}_i^{(\text{old})})^{(YX)} & (\boldsymbol{Q}_i^{(\text{old})})^{(YY)} \end{bmatrix} \begin{bmatrix} \boldsymbol{O}_i^{(XX)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}^{(YY)} \end{bmatrix} = \begin{bmatrix} \underbrace{(\boldsymbol{Q}_i^{(\text{old})})^{(XX)}\boldsymbol{O}_i^{(XX)}}_{\text{new subspace basis}} & \underbrace{(\boldsymbol{Q}_i^{(\text{old})})^{(XY)}}_{\text{unchanged basis}} \\ (\boldsymbol{Q}_i^{(\text{old})})^{(YX)}\boldsymbol{O}_i^{(XX)} & (\boldsymbol{Q}_i^{(\text{old})})^{(YY)} \end{bmatrix}. \tag{5}$$

Similarly, we update $\boldsymbol{P}_i$ in the subspace by substituting this form of $\boldsymbol{O}_i$ into eq. (3)

$$\boldsymbol{P}_i^{(\text{new})} = \boldsymbol{O}_i^\top \boldsymbol{P}_i^{(\text{old})}\boldsymbol{O}_i = \begin{bmatrix} (\boldsymbol{O}_i^{(XX)})^\top \boldsymbol{P}_i^{(XX)}\boldsymbol{O}_i^{(XX)} & (\boldsymbol{O}_i^{(XX)})^\top \boldsymbol{P}_i^{(XY)} \\ \boldsymbol{P}_i^{(YX)}\boldsymbol{O}_i^{(XX)} & \boldsymbol{P}_i^{(YY)} \end{bmatrix} \tag{6}$$

Mathematically, $\boldsymbol{Q}_i^{(\text{new})}$ remains orthogonal because it is the product of two orthogonal matrices, and $\boldsymbol{O}_i$ is orthogonal by construction. The update in eqs. (5) and (6) recovers the block-Jacobi-type update [26] when using eigen-decomposition on $\boldsymbol{P}_i^{(XX)}$ rather than QR decomposition. Thus, our reparametrization supports subspace updates via either QR or eigen-decomposition.

Inner loop and early termination Block Jacobi methods approximate a fixed matrix $\boldsymbol{P}_i$ by introducing an inner loop that repeatedly selects a block of $\boldsymbol{P}_i$, computes a local orthogonal update $\boldsymbol{O}_i$, and updates $\boldsymbol{Q}_i$ through eq. (2). Motivated by this idea, we can introduce the same K-step loop into our scheme. Unlike block Jacobi-type methods, however, $\boldsymbol{P}_i$ changes at each gradient step in Shampoo-based methods. This new setting allows us to terminate the loop early, trading approximation accuracy for runtime, since we do not need to approximate the ever-changing $\boldsymbol{P}_i$ precisely. Importantly, this reduces the cost of QR by replacing a full-basis decomposition with a small number of subspace decompositions. Empirically, using a single QR step in a subspace ($K = 1$) is often sufficient for Shampoo-based methods when the decomposition interval is small (see table 3).

Our reparametrization facilitates greedy block selections Both our approach and Jacobi-type methods use $\boldsymbol{Q}_i$ to approximate an eigenbasis of $\boldsymbol{S}_i$. Like block Jacobi-type methods, our approach requires selecting a block of $\boldsymbol{P}_i$ from which to compute a local orthogonal factor. There is no closed-form solution for selecting an optimal block, and many greedy block-selection strategies have therefore been considered in the literature. In the ideal case, $\boldsymbol{P}_i = \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i$ is diagonal. Motivated by this observation, Jacobi-type greedy methods commonly use the Frobenius norm of the off-diagonal part of $\boldsymbol{P}_i$, as a guide for selection. Because our reparametrization stores $\boldsymbol{P}_i$ explicitly, evaluating

KL-Shampoo (Old)	**KL-Shampoo (Reparameterized)**
4: Estimate eigenbasis via subspace QR (every $T \geq 1$ iterations)	
4a: Select index set $\mathbb{I}_i$ of $d_{\text{sub},i}$ columns of $\boldsymbol{P}_i$ (i.e. $\lvert\mathbb{I}_i\rvert = d_{\text{sub},i} = Bd_i$, $0 < B < 1$)	
$\boldsymbol{P}_i := \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i$ # 4 mms	Use $\boldsymbol{P}_i$ for free
$\mathbb{I}_i := \text{select}(\boldsymbol{P}_i)$	
4b: Perform subspace QR on $\boldsymbol{P}_i$	
$\boldsymbol{O}_i \leftarrow \text{qr}(\boldsymbol{P}_i[\mathbb{I}_i, \mathbb{I}_i])$ # $O(d_{\text{sub},i}^3)$	
4c: Update only the $d_{\text{sub},k}$ orthogonal bases	
$\boldsymbol{Q}_i[:, \mathbb{I}_i] \leftarrow \boldsymbol{Q}_i[:, \mathbb{I}_i]\boldsymbol{O}_i$ # 2 smms	Rotate $\boldsymbol{P}_i$ in subspace:
	$\boldsymbol{P}_i[:, \mathbb{I}_i] \leftarrow \boldsymbol{P}_i[:, \mathbb{I}_i]\boldsymbol{O}_i$ # 2 smms
	$\boldsymbol{P}_i[\mathbb{I}_i, :] \leftarrow \boldsymbol{O}_i^\top \boldsymbol{P}_i[\mathbb{I}_i, :]$ # 2 smms

Figure 3: **Step 4 with subspace QR: side-by-side comparison.** Drop-in replacement for Step 4 which uses subspace QR. The reparameterization avoids the cost of forming $\boldsymbol{P}_i$ at every QR step, since $\boldsymbol{P}_i$ stays up-to-date. **Cost.** An `smm` (subspace `mm`) costs B^2 of an `mm`.

such objectives is inexpensive, whereas doing so under the existing parametrization (i.e., storing $\boldsymbol{S}_i$) is more costly. Thus, our reparametrization makes it easy both to use existing greedy block-selection strategies and to design new greedy ones.

GPU-friendly greedy selection Existing block Jacobi-type selection methods [4, 26] require an additional loop to select a block. We instead consider a loop-free, GPU-friendly alternative: **a greedy two-phase method** inspired by the Jacobi method. In phase 1, we select the classical greedy Jacobi pair [9],

$$\text{phase 1}: \quad (k^*, j^*) = \arg\max_{k \neq j} P_{k,j}^2,$$

and in phase 2, we expand this pair into a block of size b by choosing the $b-2$ indices with the largest total coupling to (k^*, j^*),

$$\text{phase 2}: \quad S^* = \arg\max_{\substack{S \subseteq [d] \setminus \{k^*, j^*\} \\ |S| = b-2}} \sum_{x \in S} \left(P_{x,k^*}^2 + P_{x,j^*}^2 \right).$$

Here we drop the subscript of $\boldsymbol{P}_i$, b is the block size, $\mathbb{I}_i := S^* \cup \{k^*, j^*\}$ is the selected index set, and $P_{k,j}$ denotes the (k,j)-th entry of $\boldsymbol{P}_i$. This can be implemented efficiently with `torch.topk`.

Appendix D. Additional Experimental Details

We consider the following baseline training methods: SOAP [25], KL-Shampoo/SOAP [16], and Muon [17]. We train language models—nanoGPT [14] (123M) and Llama 3 [21] (119M, 313M, and 598M)—on the FineWeb dataset. We use the official implementations of SOAP [25] and KL-Shampoo/SOAP [16]. For Muon, we use the polar express implementation [1]. For each baseline, we tune all available hyperparameters, including the learning rate, weight decay, damping, β_1, and β_2, using random search over 120 runs. For Shampoo-based methods, we set the decomposition interval to $T = 10$, as suggested by Vyas et al. [25] and Lin et al. [16]. In our experiments, the

Table 4: This demonstrates how subspace selection methods supported by our reparametrization can affect the performance of KL-Shampoo (ours, BFP16, QR) on test loss. As shown below, the greedy method is preferred over the random method due to its robustness on different models.

T	B	K	**Select**	**nanoGPT (123M)**	**llama3 (119M)**
10	$1/2$	1	random	3.249	3.345
			greedy	3.247	3.344
		3	random	3.249	3.344
			greedy	3.245	3.344
		5	random	3.249	3.345
			greedy	3.244	3.345

Table 5: This demonstrates that subspace orthogonal bases of KL-Shampoo (ours, BFP16, greedy) can be updated via an eigen- or QR-decomposition, as supported by our representation. As shown below, both decomposition methods perform similarly on test loss. This motivates the use of QR, as it is lower-cost.

T	B	K	**Basis**	**nanoGPT (123M)**	**llama3 (119M)**
10	$1/2$	1	Eig	3.247	3.345
			QR	3.247	3.344
		3	Eig	3.245	3.344
			QR	3.245	3.344
		5	Eig	3.245	3.344
			QR	3.244	3.345

reparametrized methods (e.g., SOAP, KL-Shampoo, and KL-SOAP), including both full-basis and subspace variants, simply reuse the optimal hyperparameters found for the original parametrization via random search. By default, we use BFloat16 storage for each method.

We conduct five sets of experiments on four language models—nanoGPT (123M) and Llama 3 (119M, 313M, and 598M)—using the FineWeb dataset from Hugging Face. We use the default train/test split in all experiments. As strong baselines for training matrix-valued weights in neural networks, we consider Muon [17] with the polar express backend [1], SOAP [25], and KL-Shampoo/SOAP [16]. For vector-valued weights, such as those in normalization layers, we use AdamW.

In the nanoGPT experiments [14], we use a constant learning rate with linear warm-up and cool-down, a batch size of 512, and a sequence length of 1024. In all Llama 3 experiments [22], we use cosine learning-rate scheduling, a batch size of 768, and a sequence length of 512. For each method, we tune all available hyperparameters using random search with 120 runs. Our search follows a two-stage policy. In Stage 1, we explore a wider search range with 60 runs and narrow the range based on test loss. In Stage 2, we refine the search range and perform an additional 60 runs. For smaller models, namely nanoGPT (123M) and Llama 3 (119M), we train on four L40S GPUs. For larger models, namely Llama 3 (313M) and Llama 3 (598M), we train on two H100 GPUs. Due to the limited computational resources, we train each model for 10,000 iterations and report the performance of each method.

Experiment 4: Our reparametrization supports subspace updates via eigendecomposition or QR decomposition As discussed in section 2.2, our QR-based update scheme coincides with block Jacobi methods. As a result, our reparametrization also supports subspace updates via eigendecomposition. In this set of experiments, we provide empirical evidence for this claim. These experiments also support the use of QR decomposition in a subspace. From table 5, we can see that eigendecomposition and QR decomposition perform similarly. However, QR decomposition is much

faster than eigendecomposition in practice (see top right of fig. 1). This echoes the findings of Vyas et al. [25] and further motivates using QR to replace eigendecomposition, even in subspaces.

Experiment 5: Using our reparametrization reduces the runtime of Shampoo-based methods In this set of experiments, we consider (pre-)training a larger model and demonstrate the benefits of using subspace updates. From the top-left panel of fig. 1, we can see that subspace methods can outperform full-basis methods in runtime, at the cost of slightly degraded per-step performance.

Appendix E. Limitations and Future Work

In this paper, we show that our method narrows the runtime performance gap between KL-Shampoo and Muon. Unfortunately, it still does not outperform Muon in runtime. This remains a limitation of the current submission.

However, we believe that our reparametrization can further reduce runtime by combining full-basis updates with subspace updates—for example, by using cheap subspace updates together with occasional expensive full-basis updates. Another promising direction is to use the adaptive decomposition frequency suggested by Eschenhagen et al. [7]. This approach requires computing $\boldsymbol{P}_i$ to determine the update frequency automatically. Thus, our reparametrization enables an efficient implementation of this idea.

Finally, we can adapt the parallel-subspace techniques studied in the Jacobi literature and perform QR updates on multiple smaller non-overlapping subspaces in parallel to further reduce runtime. Our reparametrization naturally supports these extensions. With these improvements, we believe KL-Shampoo can be made even faster and may further close the runtime gap in future work.